Si4010 Memory Overlay Technique

1. Introduction

This application note describes the memory overlay technique of the Si4010. It also contains a detailed description of the memory overlay sample code available for users.

The Si4010 does not include a flash memory for permanent code or data storage. Instead, it contains 4.5 kB of RAM, serving as a unified CODE and XDATA, and 8 kB of NVM memory for permanent user code and data storage. The code cannot be run directly from NVM, since it is not mapped directly to the CPU address space. However, upon device reset, the device goes through a boot process during which the user application code and data—and certain factory device configurations—are copied from NVM to the RAM. As soon as the boot process finishes, the user code starts being executed from RAM at address 0x0000.

This memory structure allows users to use memory overlay technique, which is sometimes inevitable during firmware development with the Si4010. The two main causes can be the following:

- The user code and data is greater than the user-available RAM (~4 kB), thus taking advantage of the greater NVM (~8 kB) available as needed.
- Short boot time* is required. In this case, only a fraction of the entire user program can be loaded upon boot; the rest of the program can be loaded later, during runtime.

***Note:** Boot time [ms] = 2 [ms] + 3.6 [ms/kB] x NVM data [kB]

The key concept is to split up the RAM address space into two main parts. The first one is where the Main user code—Main Module in the subsequent parts of this document—will be loaded upon boot, while the second part is for loading user overlay code—Overlay Module in the subsequent parts of this document—at runtime as needed. Multiple Overlay Modules can be stored in the NVM and can be loaded to this second part of the RAM; the existing one will be overwritten whenever a new one is loaded. Additionally, there should be a separate region in the RAM address space for global variables which are used for parameter passing between the Main and Overlay Modules. In the current example, parameter passing between the Main and Overlay Modules through function parameters is not possible.

In order to make the transition between the Main Module and the actual Overlay Module simple, it is recommended to construct the user application software architecture in such a way that the interface between the two parts should be a single function call (entry function of the given Overlay Module). In other words, every Overlay Module should consist of one main function which can be called from the Main Module at runtime (after loading it). As a further option, these overlay functions may contain additional sub-functions which do not have access to any variable or function declared in the Main Module. This solution ensures the straightforward program execution flow between the Main Module and the Overlay Module.

The rest of this document describes the memory overlay technique in detail. An example of source code is provided, in which the entire user code and data is significantly less than 4 kB. However, the concept of the memory overlay technique is well demonstrated in this real user application.

The Demo Application can be run on the following development and demo boards:

- Socketed Development Board (MSC-DKPE1)
- Soldered Development Board (MSC-DKPB434)
- Key Fob Board (MSC-DAPB434)

These devices have been used for the sample code development. Testing the operation of the sample code can be done by the 4320-DAPB434 receiver board, as in the Simplified Remote Keyless Entry Demo (4010-DASKF434).

Copyright © 2010 by Silicon Laboratories

2. Sample Code Description

2.1. Functionality

The provided memory overlay sample code is the modified Key Fob demo code. If any button is pushed on the Key Fob board, it sends a packet that can be received by a demo receiver (4320-DAPB 434). A different LED combination lights up on the receiver for each button push. Also, the LED of the transmitter flashes at each transmission.

There are two separate overlay functions in the provided sample code. The first one sets up the packets to be transmitted while the second one is responsible for transmitting the packets set up previously. The Main Module monitors the buttons and calls the entry function of the Overlay Modules respectively.

2.2. Memory Map

The memory organization* of the Si4010 is similar to that of a standard 8051. Two main memory spaces are there— program memory and data memory—but there is an important difference:

The program and XDATA memory share the same address space from 0x0000 to 0x11FF in the RAM, and can be accessed via different instruction types. This unified CODE and XDATA memory structure provides the flexibility to use this memory area to CODE or XDATA as needed by the user code. This feature enables users to easily overwrite parts of the CODE memory, which is configured as XDATA during copying from NVM. The other consequence of this architecture is that the CODE and XDATA memory spaces of an application should be carefully determined, not to overlap each other in the RAM.

Overall, when setting up the memory map, the user needs to check/monitor `wBoot_DpramTrimBeg`, `wBoot_NvmUserBeg`, and `wBoot_NvmCopyAddr` system variables for determining the user-available RAM and NVM areas. Details on how to do this are provided in the next sections of this document.

These system variables are written to the chip during production test and can be viewed (read out) through the Silicon Labs IDE, by turning on the **View →Debug windows→ Si4010→ System Vars** window.

***Note:** For more information about the memory organization of the Si4010, consult the data sheet and the AN370 application note.

2.2.1. Determining the Memory Map of the RAM

During program development, after compilation, the IDE provides information about the required storage sizes in the memory areas—like DATA, XDATA, and CODE—used by the application. Based on this information, the user can locate CODE and XDATA to the desired memory areas by the appropriate linker settings. For more information, see Section “3. Linker Settings” of this document.

Figure 1 shows the memory map of the RAM and NVM of the example. In the provided sample code there are three main regions in the RAM:

1. Main Module CODE/XDATA
2. Global Variables
3. Overlay Module CODE/XDATA

The figure also shows how the CODE and the XDATA segments are configured within these main regions. The Global Variables area is accessed by both the Main Module and the Overlay Module(s). Note that for different user applications the memory map should be adjusted.

As Figure 1 shows, there is 4.5 kB RAM in the Si4010, but a small region is reserved for factory use, resulting in approximately 4 kB user-available RAM. Determining the exact number can be done by reading the value of the `wBoot_DpramTrimmBeg` system variable, which points to the first occupied address of the RAM. In other words, the user-available area is from 0x0000 – (`wBoot_DpramTrimBeg-1`).

Summary:

After boot, the Main Module will be loaded to the RAM and will never change. This Main Module program can load the Overlay Modules as needed. Loading an Overlay Module results in overwriting the existing CODE/XDATA in the user overlay part of the RAM.

2.2.2. Determining the Memory Map of the NVM

Locating the Main Module and Overlay Modules (Ovrl1, Ovrl2) in the NVM is to be accomplished via the NVM burner. For more information, see Section “5. Burning the Files” in this document.

The system variable `wBoot_NvmUserBeg` contains the address in NVM of the beginning of the user code area; therefore it is required for determining the user-available NVM area. That is from `wBoot_NvmUserBeg` – 0xFFC0. The full application code—including the Main and Overlay Modules —should fit into this range.

During the execution of the Main Module, before an Overlay Module is loaded, it is required to know the address where it is located in the NVM. This information about all Overlay Modules should be stored in the Main Module, and can be determined with the following iteration:

1. Compile each Module with the IDE.
2. Start the NVM Burner utility.
3. Add the Main Module.
4. Hit **Compose**, then the occupied NVM address space of the Main Module is provided by the NVM Burner utility. Any address after this region can be selected as a starting point of the next Module.
5. Add an Overlay Module to the address determined in the previous step.
6. Hit **Compose**, then the occupied NVM address space of the Overlay Module is provided by the NVM Burner utility. Any address after this region can be selected as a starting point of the next Module.
7. Repeat from Step 5 for each of the Overlay Modules.
8. The NVM start address of each Overlay Module can be read in the User App files table of the NVM Burner utility. These addresses should be used in the Main Module during loading of the Overlay Modules. In the current example, these addresses are stored in the `OD_main.h` header file. For more information, see Section “2.3. Program Structure” in this document.
9. After modifying the `OD_main.h` header file, recompile the Main Module.

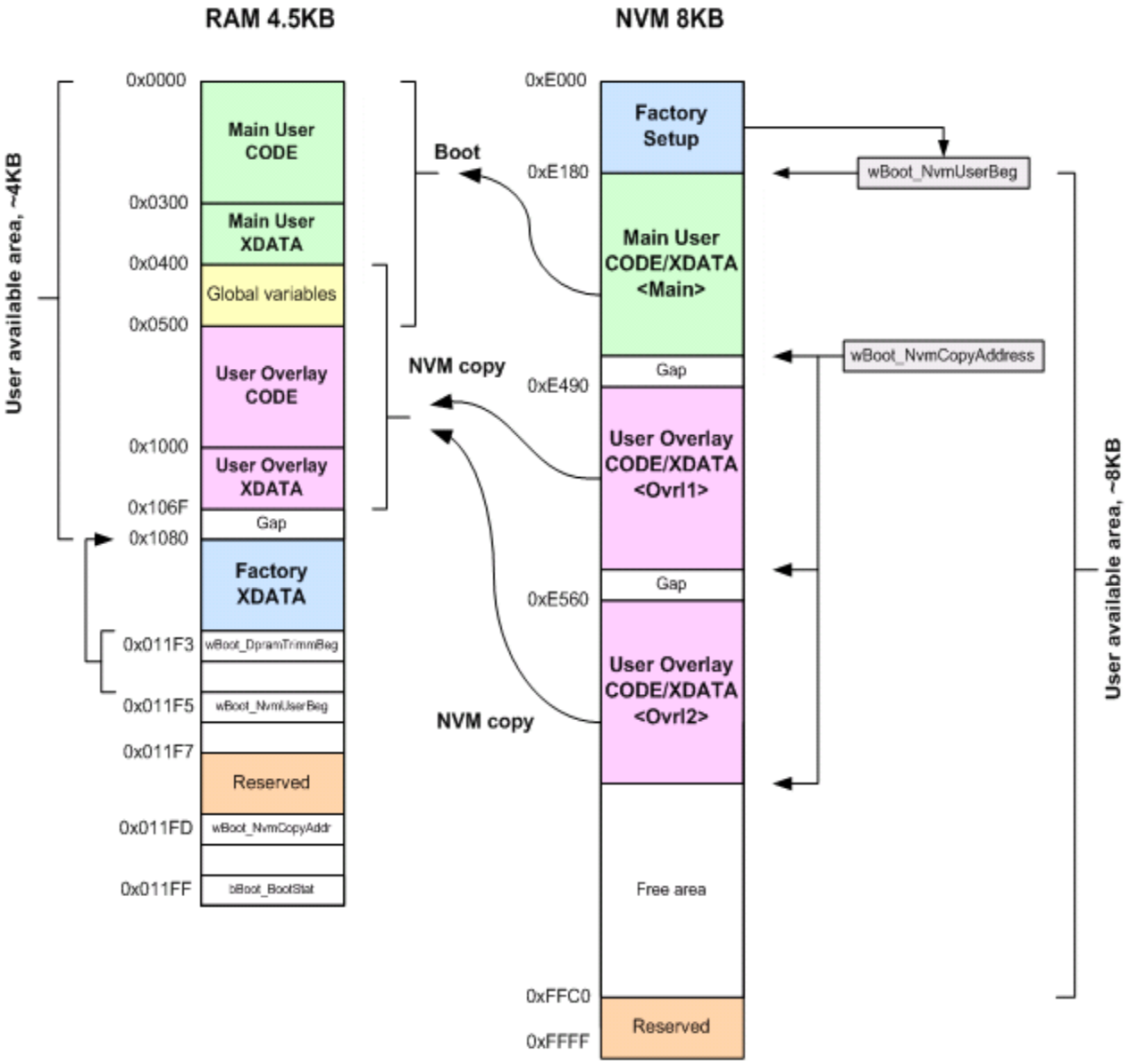

Figure 1. RAM and NVM Memory Map of the Memory Overlay Sample Code

2.2.3. Determining the Memory Map of the DATA/IDATA

The map of this memory area also should be designed carefully:

- There are reserved areas for the Si4010 API functions.
- The stack is located here.
- Separate regions are needed for each module of the application because they can have local variables here. For example, some of the API functions require a variable located in the DATA memory as an input parameter.

The IDE provides information about the memory size requirements for DATA/IDATA after the corresponding program is compiled. The DATA/IDATA memory map of the example application is shown in Figure 2. There are separate regions dedicated to the Main Module (`<Main>`), and to the Overlay Modules (`<Ovrl1>` and `<Ovrl2>`), respectively. For more information, see Section “3. Linker Settings” in this document.

The stack in the example starts at 0xA0, and is used by all of these parts. It is the user’s responsibility to avoid stack overflow,* which may result in overwriting relevant DATA/IDATA of API functions.

***Note:** A possible solution for getting the maximum stack depth can be filling it up with a special pattern, for example, with 0xFF, and check how much of it is going to be overwritten during program execution. For details, check the relevant article from the Keil website: http://www.keil.com/support/docs/192.htm.

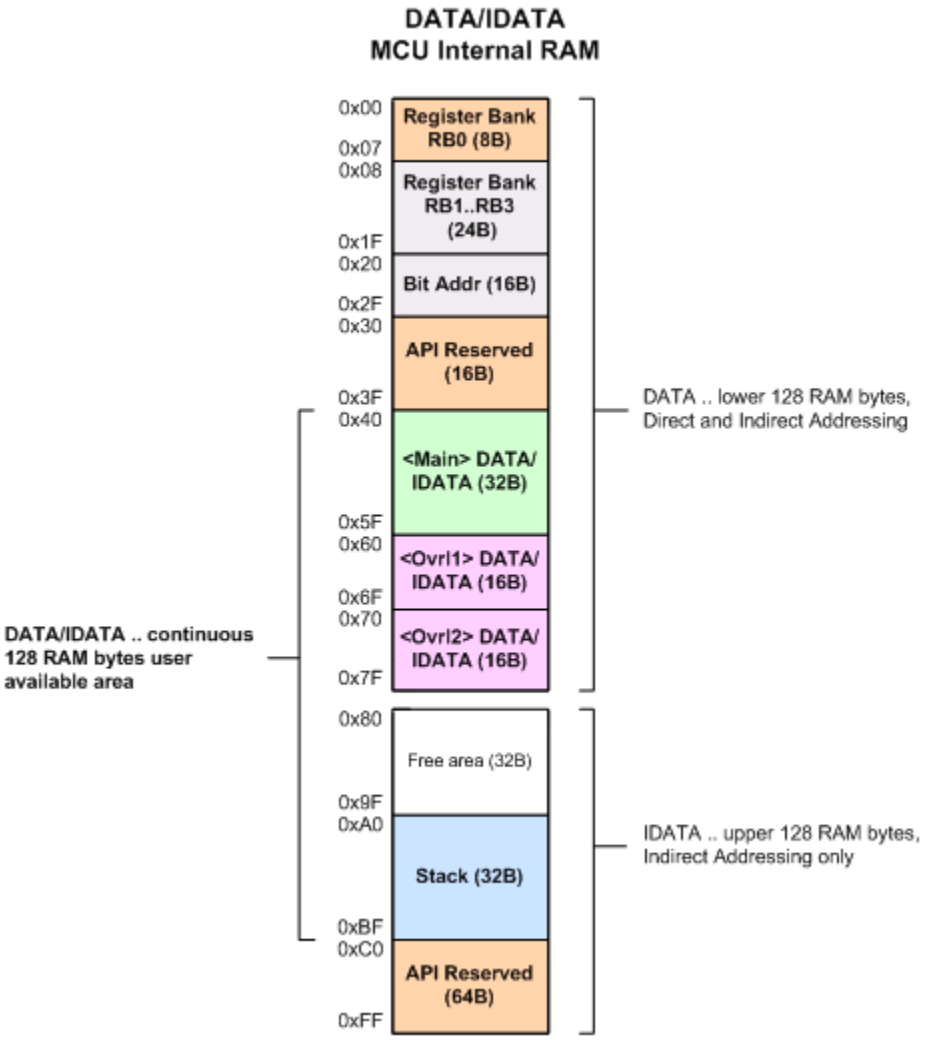

Figure 2. DATA/IDATA Memory Map of the Memory Overlay Sample Code

2.3. Program Structure

The provided sample code has three project files: `ovrl_demo_main.wsp`, `ovrl_demo_ovrl1.wsp`, and `ovrl_demo_ovrl2.wsp`. in the `../Overlay_demo/bin` folder.

Each project must be compiled and linked individually, which results in three separate IntelHEX files to be passed to the NVM burner.

From the memory overlay technique point of view, the relevant user source files of the projects are as follows:

1. `ovrl_demo_main.wsp`:
   - "`OD_main.c` —Main function of the Main Module; monitors the buttons, loads, and calls the actual overlay function
   - "`OD_main.h` —Main Module specific constants, variable and type definitions, and function prototypes
   - "`OD_global.h`—Definition of global variables and constants used for parameter passing between the Main Module and the actual Overlay user code; common header file for all the three projects
   - "`OD_M_startup.a51`—Startup file for the Main Module; the program starts being executed at 0x0000 after boot; it also sets the stack pointer
   - "`ovrl_caller.asm`—Embedded assembly code for calling overlay functions; it behaves as a subroutine in terms of saving the relevant MCU registers on to the stack
2. `ovrl_demo_ovrl1.wsp`:
   - "`OD_ovrl1_main.c` (`<Ovrl1>`)—Main function of the Overlay Module; assembles the appropriate packet to be transmitted
   - "`OD_ovrl1.h`—Overlay Module specific constants, variables, types, and function prototypes
   - "`OD_global.h`—Definition of global variables and constants used for parameter passing between the Main Module and the actual Overlay user code; common header file for all the three projects
   - "`OD_O_startup.a51`—Common startup file for the Overlay Modules; the program starts at address 0x0500 and leaves the stack as it is
3. ovrl_demo_ovrl2.wsp:
   - "`OD_ovrl2_main.c` (`<Ovrl2>`)—Main function of the Overlay Module; transmits the previously assembled packet
   - "`OD_ovrl2.h`—Overlay Module-specific constants, variables, types, and function prototypes
   - "`OD_global.h`—Definition of global variables and constants used for parameter passing between the Main Module and the actual Overlay user code; common header file for all three projects
   - "`OD_O_startup.a51`—Common startup file for the Overlay Modules; the program starts at address 0x0500 and leaves the stack as it is

2.4. Loading and Calling the Overlay Modules

Loading the Overlay Module from NVM is accomplished during runtime from the Main Module with the vOvrl_NvmLoad() API function. There are two solutions implemented in the Demo Application code for calling the entry function of the Overlay Module. The first one is via function pointers, following the principles below:

```
Part of OD_main.h:
…
typedef void (tOvrl_Func)(void);     // type definition for a function pointer
tOvrl_Func *pOvrl_Func;               // declare function pointer

Parts of OD_main.c:
…
pOvrl_Func = vOvrl_FuncReserve();// Set pointer to overlay function.
// vOvrl_FuncReserve is linked to 0x0500
…
// Load and call overlay function from NVM
wNvmAddrOvrl = wNvmAddr_PacketAssembler;
if ( vOvrl_NvmLoad(wNvmAddrOvrl) != 0xFF )
{
  #ifdef FuncPointerCall
     (pOvrl_Func)();// Call overlay function through function pointer
  #endif
  #ifdef AssemblyCall
     vOvrl_FuncCaller();// Call overlay function from assembly
  #endif
}
```

The second solution is via assembly source code using the LCALL directive, similar to using an interrupt subroutine function call. This solution also demonstrates how to save the values of the relevant MCU registers, should they be needed when modifying the sample code. The user can select between the two options by a #define directive in OD_main.h.

Since the Main Module and the Overlay Modules are separate projects in the example, and are to be linked separately, a dummy function, vOvrl_FuncReserve(), is implemented in the Main Module to have the linker set up the call tree properly. This function does nothing and will be overwritten by the first overlay function loaded. It is either used for the function pointer or for the LCALL directive.

3. Linker Settings

This application note assumes running Keil C51 for code compilation, and the BL51 linker for object linking. Therefore, the following settings might be Keil-specific, but the principles are the same with other tool chains.

All the necessary linker settings are included in the project files and can be seen in the **Project→ Tool Chain Integration→ Linker** dialog box.

Figure 3 shows the linker settings of the Main Module project file.

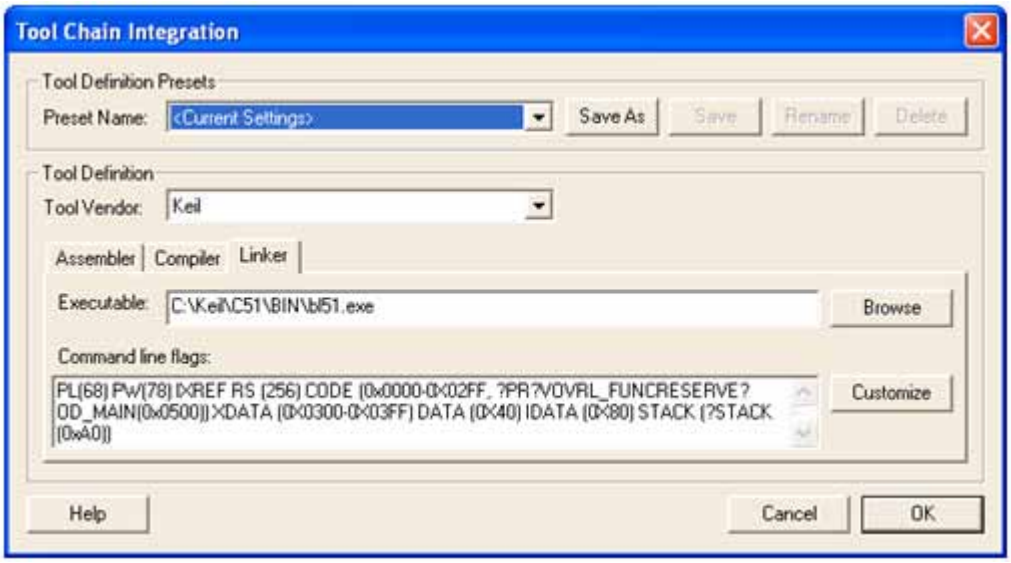

Figure 3. Linker settings for Main Module, ovrl_demo_main.wsp

The CODE resides at 0x0000-0x02FF and `v0vrl_FuncReserve()` is located at 0x0500, in the CODE area. This function is the dummy function representing entry function of the Overlay Modules. XDATA is located at 0x0300-0x03FF. Additionally, DATA, IDATA, and STACK areas are also set here. Unfortunately, with the Keil compiler it is not possible to set a range as an argument for the DATA directives in contrary to CODE and XDATA; only the start address can be set. Consequently, it is the user's responsibility to take care of avoiding overflow of these areas. The Keil uVision help and user manual provide further information on the directives used above. Only Main Module linker settings are discussed above, but the principles are the same for the Overlay Modules.

4. Checking the Call Tree

It is recommended to check the call tree whenever using function pointers in C51 compilers. In some special cases problems may abound*, which are evident immediately by checking the call tree. Keil application note APNT_129 discusses these special cases. The call tree is generated by the linker and can be found in the output file with a .M51 extension.

For the Main Module of the example, part of the call tree is shown by Figure 4.

It can be seen that `v0vrl_FuncReserve(`—which is the value of `p0vrl_Func`—is called by the `main()` of `OD_main.c` similar to the way a normal function `vAppInitial()`is called. Analyzing the call tree shows that it is correct and there is no need to manually change it.

```
SEGMENT                                              DATA_GROUP
  +--> CALLED SEGMENT                              START     LENGTH
-------------------------------------------------------------------
?C_C51STARTUP                                      -----     -----
  +--> ?PR?MAIN?OD_MAIN
  +--> ?C_INITSEG

?PR?MAIN?OD_MAIN                                   -----     -----
  +--> ?PR?VAPPINITIAL?OD_MAIN
  +--> ?PR?VOVRL_FUNCRESERVE?OD_MAIN
  +--> ?PR?VBUTTONCHECK?OD_MAIN
  +--> ?PR?_VOVRL_NVMLOAD?OD_MAIN
```

Figure 4. Call Tree Generated by the Linker for the Main Module

The `.M51` file also provides information regarding the memory usage, address of functions, variables, etc. Figure 5 contains a small part of the `.M51` file of the Overlay Module, `<Ovrl2>`. It shows that the CODE starts at address 0x0500, which is exactly what we want to achieve. When calling this overlay function from the Main Module, the program jumps to this address. The start address can be set by the startup files (`OD_M_startup.a51` and `OD_O_startup.a51`).

```
                 1200H     2E02H                *** GAP ***
BL51 BANKED LINKER/LOCATER V6.15          06/01/2010  10:10:31  PAGE 2


            XDATA   4002H     0001H     ABSOLUTE
                    4003H     0005H               *** GAP ***
            XDATA   4008H     0004H     ABSOLUTE
            XDATA   400CH     0002H     ABSOLUTE
                    400EH     0004H               *** GAP ***
            XDATA   4012H     0001H     ABSOLUTE
                    4013H     0003H               *** GAP ***
            XDATA   4016H     0001H     ABSOLUTE
            XDATA   4017H     0001H     ABSOLUTE
                    4018H     0028H               *** GAP ***
            XDATA   4040H     0010H     ABSOLUTE

            * * * * * * *   C O D E   M E M O R Y   * * * * * * *
                    0000H     0500H               *** GAP ***
            CODE    0500H     0003H     ABSOLUTE
            CODE    0503H     008CH     UNIT          ?PR?MAIN?OD_OVRL2
            CODE    058FH     0006H     UNIT          ?C_C51STARTUP
```

Figure 5. Overlay Module <Ovrl2> Starts at 0x0500

*The linker performs call tree analysis to determine if some memory areas can be safely overlaid in order to preserve as much data as possible. However, using functions pointers may result in an incorrect call tree since the linker cannot determine to which function a pointer refers. In these cases, the call tree can be fixed manually with the OVERLAY directive.

5. Burning the Files

Burning the NVM is to be done with an external software tool, the NVM Burner Utility. An application note describes it in detail; in this document only issues related to the memory overlay sample code are discussed.

The three project files result in three separate IntelHEX files. The output of the Main Module should be added to the User Boot files section, while the Overlay Modules should be added to the User App files section. Hitting the **Compose** button in the NVM burner returns the NVM address ranges for each IntelHEX file. Any address after this region can be selected as a starting point of the next Module as the NVM Address. This should be done after each HEX file is added. Figure 6 shows the settings and return values when generating the burning file for the Demo Application. Note that the NVM address ranges are the same as those described on the memory map. For more information, see Section “2.2.2. Determining the Memory Map of the NVM” in this document.

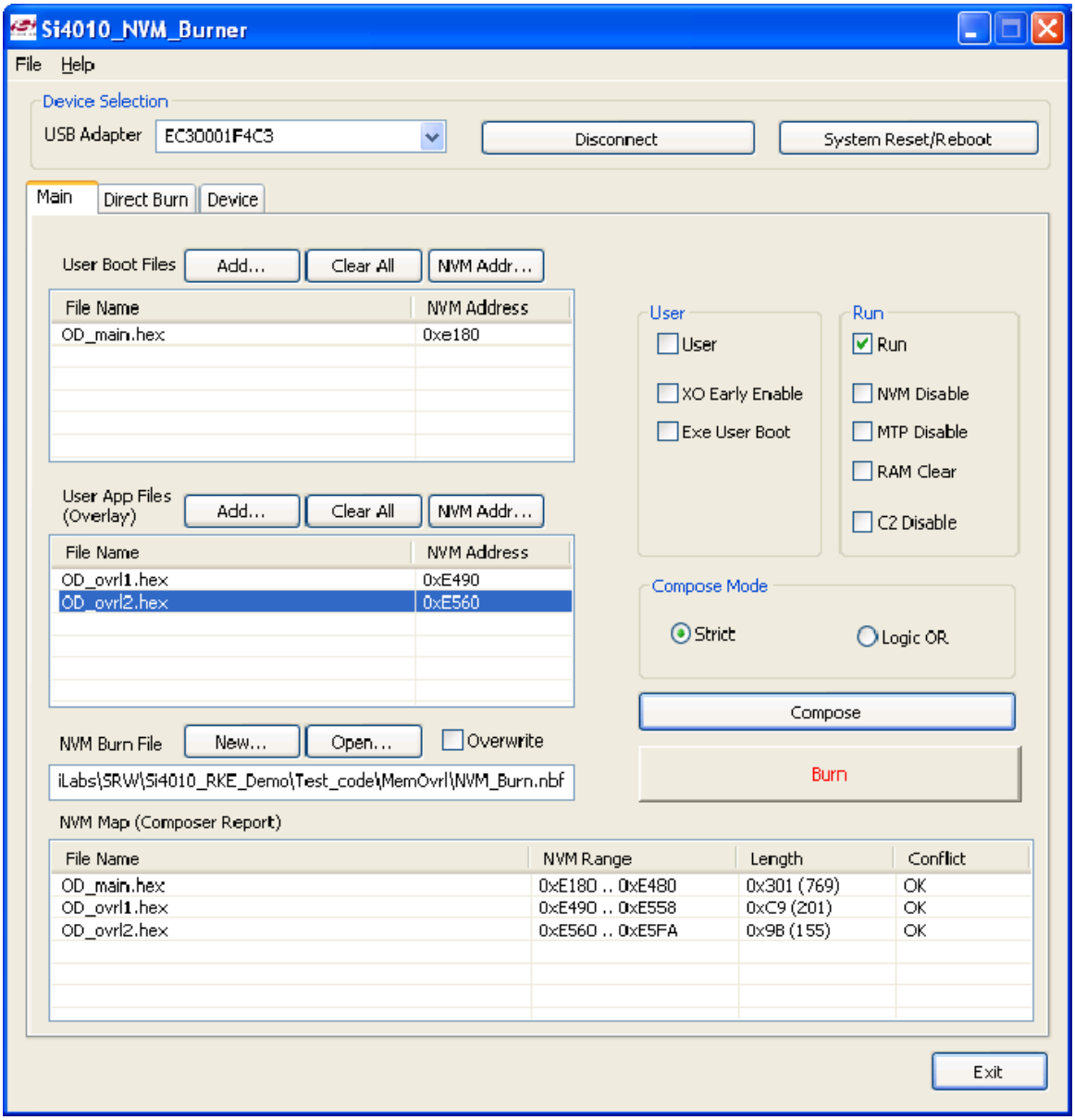

Figure 6. Burning NVM with the GUI-based NVM Burner

Note: It is necessary to set the programming level of the Si4010 before pressing the **Burn** button! If further debugging is required, then set it to User; if not, to Run. (For more information, see Section “6. Debugging the Application”.)

6. Debugging the Application

As described earlier, the sample code is partitioned into three projects: one for the Main Module and two for the Overlay Modules. These projects should be compiled and linked separately in order to get the hex files burned into the Si4010 NVM. Runtime debugging of code during software development is a very useful feature of today's development systems. Debugging applications which are built up from separate projects not linked together during development time (as in the provided sample code) is a challenging task. In the example, different levels of debugging are possible, as discussed below.

6.1. Debug of Each Module Separately

In this case, the program of any Module should be downloaded into the RAM of the Si4010 by the IDE as usual, but manual preparing of the input and output parameters of the modules is required. Since the interface between the Main and Overlay modules is done through global variables at a fixed memory location, it is easy to fill up the appropriate memory address of a given variable (which is defined in the `OD_globals.h` file) with the required test data. When debugging the Main Module, the overlay function calls should be commented out. It is also not a problem that the code of the Overlay Modules is located at a specific address because their `OD_O_startup.a51` file contains a jump from 0x0000 to the real start address of the code.

6.2. Debug of the Main Module with one of the Overlay Modules

Debugging the Overlay Modules is also possible, but only the Main plus as much of the Overlay Module as can fit into the ~4kB RAM. Each of the Overlay Modules should be compiled and linked together temporarily with the Main Module and then downloaded into the RAM. Manual preparing of the input and output parameters of the Overlay Modules may be required in this case as well.

6.3. Debug of the Main Module with all Overlay Modules

Debugging the full application firmware including the overlays on a burned part is also possible if the programming level of the Si4010 is set to User. After burning the NVM, connect the device to the IDE, and then download the output of the Main Module into it. Now it is possible to run the full code including the Overlays together with the associated copy routines. Single stepping of the code in the Overlay Module is not possible since there is no symbolic information available for the IDE about the Overlays at that time.

CONTACT INFORMATION

Silicon Laboratories Inc.

Silicon Laboratories Inc.
400 West Cesar Chavez
Austin, TX 78701
Please visit the Silicon Labs Technical Support web page:
https://www.silabs.com/support/pages/contacttechnicalsupport.aspx
and register to submit a technical support request.

The information in this document is believed to be accurate in all respects at the time of publication but is subject to change without notice. Silicon Laboratories assumes no responsibility for errors and omissions, and disclaims responsibility for any consequences resulting from the use of information included herein. Additionally, Silicon Laboratories assumes no responsibility for the functioning of undescribed features or parameters. Silicon Laboratories reserves the right to make changes without further notice. Silicon Laboratories makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Silicon Laboratories assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. Silicon Laboratories products are not designed, intended, or authorized for use in applications intended to support or sustain life, or for any other application in which the failure of the Silicon Laboratories product could create a situation where personal injury or death may occur. Should Buyer purchase or use Silicon Laboratories products for any such unintended or unauthorized application, Buyer shall indemnify and hold Silicon Laboratories harmless against all claims and damages.

Silicon Laboratories and Silicon Labs are trademarks of Silicon Laboratories Inc.
Other products or brandnames mentioned herein are trademarks or registered trademarks of their respective holders